# In-Situ GPU Performance Analysis: A Technical Guide to Programmatic Counter Collection with the rocprofiler-sdk API

## Section 1: A Technical Primer: The rocprofiler-sdk "Tool" Architecture

### 1.1 Introduction: Beyond rocprofv3

The AMD ROCm (Radeon Open Compute) platform provides a command-line interface, rocprofv3, for application tracing and hardware counter collection.1 While this tool is effective for general-purpose profiling, it functions as a high-level wrapper around the rocprofiler-sdk, a sophisticated C/C++ library for in-depth performance analysis.3 For developers and researchers requiring fine-grained, conditional, or in-situ monitoring—such as enabling counters only during specific application phases or for designated kernel executions—direct programmatic access to the rocprofiler-sdk API is the required and only viable method.

This report provides a comprehensive technical guide and a complete, self-contained code sample for instrumenting a HIP (Heterogeneous-Compute Interface for Portability) application using the rocprofiler-sdk C++ API directly. The objective is to construct a custom profiling "tool" capable of extracting and displaying GPU hardware performance counters.

The solution presented here involves building a shared library (.so) that functions as a *plugin* to the ROCm runtime, rather than a traditional application that links against a profiler library. This architectural distinction is fundamental to the design of the rocprofiler-sdk and is the primary concept that must be mastered to leverage its capabilities.

### 1.2 The Core Paradigm: Inversion of Control

The rocprofiler-sdk employs an *inversion of control* (IoC) architecture for tool loading and initialization. In a traditional profiling model, an application's main() function would explicitly initialize the profiler, call functions to start and stop data collection, and then retrieve the results. The rocprofiler-sdk model inverts this relationship:

1. The target application (e.g., a HIP program) is launched.
2. During the initialization of the ROCm runtimes (HIP and HSA), a helper library, rocprofiler-register, is loaded.6
3. This helper library scans the application's address space for a specific, publicly-visible C symbol: rocprofiler\_configure.4
4. If this symbol is found, the rocprofiler-register library invokes it, passing control to the custom tool.
5. The tool, from within its rocprofiler\_configure function, returns a struct containing function pointers to its *actual* initialization and finalization routines.8

This design represents a significant architectural shift from legacy ROCm profilers (ROCProfiler v1/v2 and ROCTracer), which often relied on environment variables like HSA\_TOOLS\_LIB or ROCP\_TOOL\_LIB.9 The documentation for the new SDK describes this older method as a "race" to set variables before the runtime fully initialized, a design that was prone to failure, especially in complex environments.4

The new symbol-discovery model is a "drastic improvement" 4 that ensures the tool is loaded and initialized at a stable, well-defined point in the runtime's startup sequence. This mechanism is inherently more robust and is designed to solve the thread-safety and initialization-order problems that plagued earlier tools, particularly in multi-library contexts like MPI or Python-based frameworks.3 The developer of a custom tool must therefore embrace this passive, plugin-based model.

### 1.3 The Tool Loading and Registration Workflow

To activate a custom tool, the ROCm runtime must be instructed to load its shared library into the application's address space, making the rocprofiler\_configure symbol discoverable.

* **The rocprofiler-register Library:** This helper library is the core mediator. It is loaded by the HIP and HSA runtimes and is responsible for finding and initializing all rocprofiler-sdk tools.6
* **The rocprofiler\_configure Entry Point:** This is the single, mandatory C-style function that a custom tool must export. The rocprofiler-sdk/registration.h header file provides the necessary extern "C" linkage and compiler visibility attributes (\_\_attribute\_\_((visibility("default")))) to ensure this symbol is properly exported from the shared library and found by rocprofiler-register.4
* **Tool Activation:** While standard Linux mechanisms like LD\_PRELOAD can be used to inject the tool's shared library 2, the correct, SDK-native method is to use the ROCP\_TOOL\_LIBRARIES environment variable.4 Setting this variable (e.g., export ROCP\_TOOL\_LIBRARIES=./libmytool.so) provides a list of paths for rocprofiler-register to search, load, and scan for the rocprofiler\_configure symbol.

## Section 2: The Tool Lifecycle: Initialization, State Management, and Finalization

### 2.1 The Handshake: Implementing rocprofiler\_configure

The rocprofiler\_configure function is the handshake between the ROCm runtime and the custom tool. Its implementation is typically boilerplate code that:

1. Receives versioning information and a client\_id pointer from the SDK.8
2. Optionally sets a human-readable name for the tool (e.g., client\_id->name = "MyCounterTool").8
3. Initializes a persistent state object for the tool.
4. Returns a pointer to a static rocprofiler\_tool\_configure\_result\_t struct.8

This rocprofiler\_tool\_configure\_result\_t struct is defined in the SDK and must be populated with:

* struct\_size: The size of the struct, for versioning (e.g., sizeof(rocprofiler\_tool\_configure\_result\_t)).
* initialize: A function pointer to the tool's *actual* initialization routine (e.g., tool\_init).
* finalize: A function pointer to the tool's finalization routine (e.g., tool\_fini).
* tool\_data: A void\* pointer to the persistent state object created in step 3.8

### 2.2 The "Real" Main: tool\_init(rocprofiler\_client\_finalize\_t fini\_func, void\* tool\_data)

The function specified in the initialize field (e.g., tool\_init) serves as the tool's true entry point. It is called by the SDK after all tools have been configured.8 This function receives two crucial arguments:

* fini\_func: A function pointer of type rocprofiler\_client\_finalize\_t. The tool can store and later call this function to programmatically request its own shutdown and trigger the tool\_fini callback.8
* tool\_data: The *exact same* void\* pointer that was provided in the rocprofiler\_tool\_configure\_result\_t struct.8

This tool\_data pointer is the lynchpin for managing state in a C++ tool that must interface with a pure C API. The API uses static callback functions (e.g., for buffer flushing and dispatch interception) which have no concept of C++ objects. The tool\_data pointer provides the standard C-style "pass-through" mechanism to bridge this gap.

The required implementation pattern is:

1. In rocprofiler\_configure, new a C++ class or struct (e.g., tool\_state\_t) to hold all tool state (context IDs, buffer IDs, profile configurations, maps, mutexes, etc.).
2. Cast this object pointer to void\* and assign it to the tool\_data field of the returned struct.
3. In tool\_init, and every subsequent callback function, static\_cast this void\* pointer *back* to a tool\_state\_t\*. This provides access to the persistent, object-oriented state from within the static C callbacks.

### 2.3 The Critical Deadlock: What You ***CANNOT*** Do in tool\_init

A critical, non-obvious limitation exists for the tool\_init function. The official documentation explicitly warns:

"ROCprofiler-SDK explicitly **does NOT support** calls to any runtime function (HSA, HIP, and so on) during tool initialization. Invoking any functions from the runtimes during this phase will result in a **deadlock**." 3

This behavior is a direct and logical consequence of the inversion-of-control architecture. The rocprofiler-register library loads and calls the tool's rocprofiler\_configure and tool\_init functions *during* the runtime's own initialization sequence.6 At this early stage, the HIP and HSA runtimes are not fully operational.

If the code within tool\_init attempts to call any HIP API (e.g., hipGetDeviceCount() or hipGetAgent()), it is calling into a library that is incomplete and likely blocked, waiting for the very initialization thread that is now executing tool\_init. This creates a classic re-entrancy deadlock.

Therefore, tool\_init must *only* be used for:

1. Calling rocprofiler-sdk API functions (e.g., rocprofiler\_create\_context).
2. Setting up the tool's internal state.

All interactions with HIP/HSA agents and other runtime features must be deferred to later callbacks that are guaranteed to execute after the runtime is fully initialized.

### 2.4 Graceful Shutdown: tool\_fini(void\* tool\_data)

The tool\_fini function, pointed to by the finalize member of the configuration struct, serves as the tool's destructor. It is invoked automatically by the SDK when the application exits (via an atexit handler) or when the tool itself calls the fini\_func pointer provided in tool\_init.8

This function's sole responsibility is to clean up all resources allocated by the tool. It receives the tool\_data pointer one last time, which must be static\_cast back to the state object. Typical cleanup operations include:

* Stopping any active profiling contexts (e.g., rocprofiler\_stop\_context).
* Destroying profiles, buffers, and contexts (e.g., rocprofiler\_destroy\_counter\_config, rocprofiler\_destroy\_buffer, rocprofiler\_destroy\_context).
* Finally, deleteing the tool\_data state object itself to free its memory.

## Section 3: The Counter Collection Service Workflow: A Step-by-Step API Guide

### 3.1 Dispatch Counting vs. Device Counting: A Critical Distinction

The rocprofiler-sdk provides two distinct modes for hardware counter collection 16:

1. **Device Counting:** Collects device-level counters over a specified time range, not tied to any specific kernel.
2. **Dispatch Counting:** Collects counters on a *per-kernel-dispatch* basis.

For the user's goal of "instrumenting a short GPU program," Dispatch Counting is the correct mode. However, this mode has a profound and critical performance implication: **it serializes kernel execution**.

The documentation states, "Note that dispatch counting allows only a single kernel to execute in hardware at a time" 16 and "Counter collection in dispatch counting mode requires serialized execution of kernels...".16

This "Heisenberg effect"—where the act of measurement fundamentally alters the system's behavior—is a necessary trade-off. Hardware performance counters are a finite, global resource on the GPU. To accurately attribute counter values (e.g., L2 cache hits) to a *single* kernel dispatch, the SDK must prevent any other kernels from running concurrently and "polluting" the counter registers.

This serialization has two major consequences:

1. All application-level concurrency (e.g., multiple streams, asynchronous execution) will be nullified, and the application's total runtime will be significantly elongated.
2. Applications that rely on co-dependent kernels (i.e., two kernels that *must* execute simultaneously on the same device) will experience a deadlock when profiled in this mode.16

### 3.2 The API Workflow (as performed in tool\_init)

The following sequence of API calls, synthesized from the rocprofiler-sdk documentation 16, outlines the complete procedure for setting up the dispatch counting service. This entire workflow must be executed from within the tool\_init function.

Step 1: Create a Context

A context is the top-level object that "bundles" one or more profiling services.12

C++

rocprofiler\_context\_id\_t ctx;  
ROCPROFILER\_CALL(rocprofiler\_create\_context(&ctx), "context creation failed");

Step 2: Create a Buffer

A buffer is created and bound to the context. This is where the SDK will asynchronously write profiling data (headers, counter values, etc.). A buffered\_callback function must be provided, which the SDK will invoke in a separate thread when the buffer is full or flushed.

C++

rocprofiler\_buffer\_id\_t buff;  
ROCPROFILER\_CALL(rocprofiler\_create\_buffer(ctx, 4096, 2048,  
 ROCPROFILER\_BUFFER\_POLICY\_LOSSLESS,  
 buffered\_callback, // Our callback function  
 my\_tool\_state, // Pass-through data  
 &buff), "buffer creation failed");

Step 3: Find the Target GPU Agent

Due to the deadlock limitation (Section 2.3), HIP/HSA APIs cannot be used here. Instead, the SDK provides its own agent query functions.

C++

// This requires a simple iteration callback to find the first GPU  
rocprofiler\_agent\_id\_t gpu\_agent;  
ROCPROFILER\_CALL(rocprofiler\_query\_available\_agents(ROCPROFILER\_AGENT\_INFO\_VERSION\_0,  
 agent\_iterate\_cb, // A callback  
 sizeof(rocprofiler\_agent\_t),  
 &gpu\_agent), // Output  
 "query available agents failed");

Step 4: Discover and Select Counters

Query the selected agent for its list of supported hardware counters. This is used to find the unique rocprofiler\_counter\_id\_t for each counter name (e.g., "SQ\_WAVES").

C++

// This also uses an iteration callback  
std::vector<rocprofiler\_counter\_id\_t> counters\_to\_collect;  
ROCPROFILER\_CALL(rocprofiler\_iterate\_agent\_supported\_counters(gpu\_agent,  
 counter\_iterate\_cb,  
 &counters\_to\_collect),  
 "Could not fetch supported counters");

Step 5: Build the Counter Profile

The list of desired counter IDs is "compiled" into an immutable, agent-specific profile object.

C++

rocprofiler\_counter\_config\_id\_t profile;  
ROCPROFILER\_CALL(rocprofiler\_create\_counter\_config(gpu\_agent,  
 counters\_to\_collect.data(),  
 counters\_to\_collect.size(),  
 &profile),  
 "Could not construct profile");

Step 6: Configure the Dispatch Counting Service

This key function binds the context, buffer, and profile together. It also registers the dispatch\_callback function.

C++

ROCPROFILER\_CALL(rocprofiler\_configure\_buffer\_dispatch\_counting\_service(  
 ctx, buff,  
 dispatch\_callback, // Our callback function  
 my\_tool\_state), // Pass-through data  
 "Could not setup dispatch counting service");

Step 7: Activate the Context

Finally, the context is started. This "arms" the profiler, which will now begin intercepting kernel dispatches.

C++

ROCPROFILER\_CALL(rocprofiler\_start\_context(ctx), "context start failed");

## Section 4: A Complete, Annotated C++ Implementation

The following is a complete, self-contained, and annotated code sample. It is split into two files: main.cpp (the target HIP application) and tool.cpp (the custom profiler tool). This sample is designed to be a functional replacement for the examples noted as "unavailable" in the documentation.3

### 4.1 Part 1: The Target Application (main.cpp)

This is a minimal HIP application that launches a simple vector\_add kernel. The profiler tool will instrument this kernel.

C++

// main.cpp  
// A simple HIP application to be profiled.  
#**include** <hip/hip\_runtime.h>  
#**include** <iostream>  
#**include** <thread>  
#**include** <chrono>  
  
#**define** CHECK(cmd) \  
{ \  
 hipError\_t err = cmd; \  
 **if** (err!= hipSuccess) { \  
 std::cerr << "HIP Error: " << hipGetErrorString(err) \  
 << " at " << \_\_FILE\_\_ << ":" << \_\_LINE\_\_ << std::endl; \  
 exit(EXIT\_FAILURE); \  
 } \  
}  
  
// Simple vector addition kernel  
\_\_global\_\_ void vector\_add(const float\* a, const float\* b, float\* c, int n)  
{  
 int idx = blockIdx.x \* blockDim.x + threadIdx.x;  
 if (idx < n) {  
 c[idx] = a[idx] + b[idx];  
 }  
}  
  
int main()  
{  
 const int n = 1024 \* 1024;  
 const size\_t bytes = n \* sizeof(float);  
  
 float \*h\_a, \*h\_b, \*h\_c;  
 float \*d\_a, \*d\_b, \*d\_c;  
  
 // Allocate host memory  
 h\_a = new float[n];  
 h\_b = new float[n];  
 h\_c = new float[n];  
  
 // Initialize host vectors  
 for (int i = 0; i < n; ++i) {  
 h\_a[i] = 1.0f;  
 h\_b[i] = 2.0f;  
 }  
  
 // Allocate device memory  
 CHECK(hipMalloc(&d\_a, bytes));  
 CHECK(hipMalloc(&d\_b, bytes));  
 CHECK(hipMalloc(&d\_c, bytes));  
  
 // Copy data from host to device  
 CHECK(hipMemcpy(d\_a, h\_a, bytes, hipMemcpyHostToDevice));  
 CHECK(hipMemcpy(d\_b, h\_b, bytes, hipMemcpyHostToDevice));  
  
 const int threads\_per\_block = 256;  
 const int blocks = (n + threads\_per\_block - 1) / threads\_per\_block;  
  
 std::cout << "[App] Launching vector\_add kernel..." << std::endl;  
  
 // Launch the kernel  
 hipLaunchKernelGGL(vector\_add, dim3(blocks), dim3(threads\_per\_block), 0, 0,  
 d\_a, d\_b, d\_c, n);  
  
 CHECK(hipDeviceSynchronize());  
 std::cout << "[App] Kernel launch complete." << std::endl;  
  
 // Copy result back from device to host  
 CHECK(hipMemcpy(h\_c, d\_c, bytes, hipMemcpyDeviceToHost));  
  
 // Verify result  
 int errors = 0;  
 for (int i = 0; i < n; ++i) {  
 if (h\_c[i]!= 3.0f) {  
 errors++;  
 }  
 }  
 std::cout << "[App] Verification: " << (errors == 0? "PASSED" : "FAILED") << std::endl;  
  
 // Clean up  
 delete h\_a;  
 delete h\_b;  
 delete h\_c;  
 CHECK(hipFree(d\_a));  
 CHECK(hipFree(d\_b));  
 CHECK(hipFree(d\_c));  
  
 // IMPORTANT: Wait for a moment before exiting.  
 // The rocprofiler buffer flush (buffered\_callback) is asynchronous.  
 // If main() exits too quickly, the callback may not have time to be  
 // invoked and print the counter data.  
 std::cout << "[App] Shutting down in 1 second..." << std::endl;  
 std::this\_thread::sleep\_for(std::chrono::seconds(1));  
  
 return 0;  
}

### 4.2 Part 2: The Profiler Tool (tool.cpp)

This is the shared library that implements the rocprofiler-sdk tool API. It finds the GPU, sets up the counters, and defines the callbacks to print the data.

C++

// tool.cpp  
// The custom rocprofiler-sdk tool, to be compiled as libmytool.so  
#**include** <rocprofiler-sdk/rocprofiler.h>  
#**include** <rocprofiler-sdk/registration.h>  
  
#**include** <cassert>  
#**include** <cstdint>  
#**include** <iostream>  
#**include** <mutex>  
#**include** <string>  
#**include** <vector>  
#**include** <map>  
#**include** <sstream>  
  
// Helper macro for checking rocprofiler-sdk status  
#**define** ROCPROFILER\_CALL(call, msg) \  
 { \  
 rocprofiler\_status\_t status = call; \  
 **if** (status!= ROCPROFILER\_STATUS\_SUCCESS) { \  
 std::cerr << " ERROR: " << msg << " failed with status " \  
 << status << std::endl; \  
 std::abort(); \  
 } \  
 }  
  
// --- Global State and Callbacks ------------------------------------------- //  
  
// We will request these counters.  
// Note: TCC\_HIT is an instanced counter. We must specify a specific  
// instance (e.g., TCC\_HIT) or a summed metric (TCC\_HIT\_sum).  
// Using the base name "TCC\_HIT" will fail.  
std::vector<const char\*> REQ\_COUNTER\_NAMES = {"SQ\_WAVES", "TCC\_HIT"};  
  
// Thread-safe printing  
std::mutex g\_print\_mutex;  
  
// This struct holds all persistent state for our tool.  
// A pointer to this object will be passed as tool\_data.  
struct tool\_state\_t  
{  
 rocprofiler\_client\_id\_t\* client\_id = nullptr;  
 rocprofiler\_client\_finalize\_t client\_fini = nullptr;  
 rocprofiler\_context\_id\_t ctx = {0};  
 rocprofiler\_buffer\_id\_t buff = {0};  
 rocprofiler\_agent\_id\_t gpu\_agent = {0};  
 rocprofiler\_counter\_config\_id\_t profile = {0};  
  
 // We will build a map to resolve counter IDs to names in the buffer callback  
 std::map<uint64\_t, std::string> counter\_id\_to\_name;  
};  
  
  
/\*\*  
 \* @brief The "dispatch callback".  
 \* This function is called by the SDK \*before\* a kernel is dispatched.  
 \* It \*asks\* our tool what configuration to use for this \*specific\* dispatch.  
 \*/  
void dispatch\_callback(  
 rocprofiler\_dispatch\_counting\_service\_data\_t dispatch\_data,  
 rocprofiler\_counter\_config\_id\_t\* config,  
 rocprofiler\_user\_data\_t\* user\_data,  
 void\* callback\_data)  
{  
 (void) callback\_data; // Unused  
  
 // Retrieve our persistent state  
 tool\_state\_t\* state = static\_cast<tool\_state\_t\*>(user\_data->value);  
  
 // Unconditionally apply our profile to this dispatch.  
 // A more advanced tool could inspect dispatch\_data.kernel\_name  
 // and decide \*not\* to profile by setting \*config = {0}.  
 \*config = state->profile;  
  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " Dispatch Callback: Intercepted kernel '"  
 << dispatch\_data.kernel\_name << "'. Applying counter profile."  
 << std::endl;  
}  
  
  
/\*\*  
 \* @brief The "buffered callback".  
 \* This function is called by the SDK in a separate thread when the  
 \* internal buffer is full or flushed at context stop.[16, 18]  
 \* It provides the raw data records.  
 \*/  
void buffered\_callback(rocprofiler\_context\_id\_t context,  
 rocprofiler\_buffer\_id\_t buffer,  
 rocprofiler\_record\_header\_t\*\* headers,  
 size\_t num\_headers,  
 void\* user\_data,  
 uint64\_t drop\_count)  
{  
 (void) context; (void) buffer; (void) drop\_count; // Unused  
  
 // Retrieve our persistent state  
 tool\_state\_t\* state = static\_cast<tool\_state\_t\*>(user\_data);  
  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << "\n BUFFER CALLBACK: Received " << num\_headers  
 << " records." << std::endl;  
  
 // Iterate over all records in the buffer [15, 20]  
 for (size\_t i = 0; i < num\_headers; ++i)  
 {  
 rocprofiler\_record\_header\_t\* header = headers[i];  
  
 if (header->category == ROCPROFILER\_BUFFER\_CATEGORY\_COUNTERS)  
 {  
 // Switch on the specific record kind  
 switch(header->kind)  
 {  
 // This header provides metadata for the dispatch  
 case ROCPROFILER\_COUNTER\_RECORD\_PROFILE\_COUNTING\_DISPATCH\_HEADER:  
 {  
 auto\* record = static\_cast<  
 rocprofiler\_dispatch\_counting\_service\_record\_t\*>(header->payload);  
 std::cout << " > DISPATCH HEADER:\n"  
 << " Correlation ID: " << record->correlation\_id.internal  
 << "\n Kernel Name: " << record->kernel\_name  
 << std::endl;  
 break;  
 }  
 // This record contains an actual counter value  
 case ROCPROFILER\_COUNTER\_RECORD\_VALUE:  
 {  
 auto\* record = static\_cast<rocprofiler\_counter\_record\_t\*>(  
 header->payload);  
  
 // Use our map to find the counter name from its ID  
 std::string name = "UNKNOWN";  
 auto it = state->counter\_id\_to\_name.find(record->counter\_id.handle);  
 if (it!= state->counter\_id\_to\_name.end()) {  
 name = it->second;  
 }  
  
 std::cout << " > COUNTER VALUE:\n"  
 << " Counter ID: " << record->counter\_id.handle  
 << "\n Name: " << name  
 << "\n Value: " << record->value  
 << std::endl;  
 break;  
 }  
 default:  
 std::cout << " > Other Counter Record (Kind: "  
 << header->kind << ")" << std::endl;  
 break;  
 }  
 }  
 }  
 std::cout << " BUFFER CALLBACK: Finished processing records.\n" << std::endl;  
}  
  
  
// --- Agent and Counter Iteration Callbacks -------------------------------- //  
  
/\*\*  
 \* @brief Callback for rocprofiler\_query\_available\_agents.  
 \* Finds the first available GPU agent.  
 \*/  
rocprofiler\_status\_t agent\_iterate\_cb(rocprofiler\_agent\_version\_t version,  
 const void\*\* agents,  
 size\_t num\_agents,  
 void\* user\_data)  
{  
 (void) version;  
 assert(version == ROCPROFILER\_AGENT\_INFO\_VERSION\_0);  
  
 rocprofiler\_agent\_id\_t\* out\_agent =  
 static\_cast<rocprofiler\_agent\_id\_t\*>(user\_data);  
  
 for (size\_t i = 0; i < num\_agents; ++i)  
 {  
 auto\* agent = static\_cast<const rocprofiler\_agent\_v0\_t\*>(agents[i]);  
 if (agent->type == ROCPROFILER\_AGENT\_TYPE\_GPU)  
 {  
 \*out\_agent = agent->id;  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " Found GPU agent: " << agent->name  
 << " (ID: " << agent->id.handle << ")" << std::endl;  
 return ROCPROFILER\_STATUS\_SUCCESS\_STOP; // Stop iterating  
 }  
 }  
 return ROCPROFILER\_STATUS\_SUCCESS;  
}  
  
/\*\*  
 \* @brief Callback for rocprofiler\_iterate\_agent\_supported\_counters.  
 \* Finds the counters specified in REQ\_COUNTER\_NAMES and populates  
 \* the tool\_state\_t with their IDs and names.  
 \*/  
rocprofiler\_status\_t counter\_iterate\_cb(rocprofiler\_agent\_id\_t agent\_id,  
 rocprofiler\_counter\_info\_v0\_t\* counters,  
 size\_t num\_counters,  
 void\* user\_data)  
{  
 (void) agent\_id;  
 tool\_state\_t\* state = static\_cast<tool\_state\_t\*>(user\_data);  
  
 for (size\_t i = 0; i < num\_counters; ++i)  
 {  
 for (const char\* req\_name : REQ\_COUNTER\_NAMES)  
 {  
 if (strcmp(counters[i].name, req\_name) == 0)  
 {  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " Found requested counter: " << counters[i].name  
 << " (ID: " << counters[i].id.handle << ")" << std::endl;  
  
 // Store the ID for building the profile  
 state->profile\_counters.push\_back(counters[i].id);  
  
 // Store the ID->Name mapping for the buffer callback  
 state->counter\_id\_to\_name[counters[i].id.handle] = counters[i].name;  
 }  
 }  
 }  
 return ROCPROFILER\_STATUS\_SUCCESS;  
}  
  
  
// --- Tool Initialization and Finalization --------------------------------- //  
  
/\*\*  
 \* @brief The "real" tool initialization function.  
 \* This is called by the SDK after rocprofiler\_configure.  
 \*/  
int tool\_init(rocprofiler\_client\_finalize\_t fini\_func, void\* tool\_data)  
{  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " tool\_init starting..." << std::endl;  
  
 tool\_state\_t\* state = static\_cast<tool\_state\_t\*>(tool\_data);  
 state->client\_fini = fini\_func;  
  
 // --- Start of Workflow from Section 3.2 ---  
  
 // Step 1: Create a Context  
 ROCPROFILER\_CALL(rocprofiler\_create\_context(&state->ctx),  
 "context creation failed");  
  
 // Step 2: Create a Buffer  
 ROCPROFILER\_CALL(rocprofiler\_create\_buffer(state->ctx, 4096, 2048,  
 ROCPROFILER\_BUFFER\_POLICY\_LOSSLESS,  
 buffered\_callback,  
 state, // Pass our state  
 &state->buff),  
 "buffer creation failed");  
  
 // Step 3: Find the Target GPU Agent  
 ROCPROFILER\_CALL(rocprofiler\_query\_available\_agents(  
 ROCPROFILER\_AGENT\_INFO\_VERSION\_0,  
 agent\_iterate\_cb,  
 sizeof(rocprofiler\_agent\_t),  
 &state->gpu\_agent),  
 "query available agents failed");  
  
 if (state->gpu\_agent.handle == 0) {  
 std::cerr << " ERROR: No GPU agent found." << std::endl;  
 return -1;  
 }  
  
 // Step 4: Discover and Select Counters  
 std::vector<rocprofiler\_counter\_id\_t> counter\_ids;  
 struct counter\_query\_data {  
 tool\_state\_t\* state\_ptr;  
 std::vector<rocprofiler\_counter\_id\_t>\* ids\_ptr;  
 } query\_data = {state, &counter\_ids};  
   
 // Custom iteration callback to find all counters  
 auto find\_counters\_cb =  
 (rocprofiler\_agent\_id\_t,  
 rocprofiler\_counter\_info\_v0\_t\* counters,  
 size\_t num\_counters,  
 void\* user\_data)  
 {  
 auto\* data = static\_cast<counter\_query\_data\*>(user\_data);  
 for (size\_t i = 0; i < num\_counters; ++i)  
 {  
 for (const char\* req\_name : REQ\_COUNTER\_NAMES)  
 {  
 if (strcmp(counters[i].name, req\_name) == 0)  
 {  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " Found requested counter: "  
 << counters[i].name << std::endl;  
 data->ids\_ptr->push\_back(counters[i].id);  
 data->state\_ptr->counter\_id\_to\_name[counters[i].id.handle] =  
 counters[i].name;  
 }  
 }  
 }  
 return ROCPROFILER\_STATUS\_SUCCESS;  
 };  
  
 ROCPROFILER\_CALL(rocprofiler\_iterate\_agent\_supported\_counters(  
 state->gpu\_agent,  
 find\_counters\_cb,  
 &query\_data),  
 "Could not fetch supported counters");  
  
 if(counter\_ids.empty()) {  
 std::cerr << " ERROR: Could not find any of the requested counters."  
 << std::endl;  
 return -1;  
 }  
  
 // Step 5: Build the Counter Profile  
 ROCPROFILER\_CALL(rocprofiler\_create\_counter\_config(  
 state->gpu\_agent,  
 counter\_ids.data(),  
 counter\_ids.size(),  
 &state->profile),  
 "Could not construct profile");  
  
 // Step 6: Configure the Dispatch Counting Service  
 ROCPROFILER\_CALL(rocprofiler\_configure\_buffer\_dispatch\_counting\_service(  
 state->ctx, state->buff,  
 dispatch\_callback,  
 static\_cast<void\*>(state)), // Pass our state  
 "Could not setup dispatch counting service");  
  
 // Step 7: Activate the Context  
 ROCPROFILER\_CALL(rocprofiler\_start\_context(state->ctx),  
 "context start failed");  
  
 // --- End of Workflow ---  
  
 std::cout << " tool\_init successful. Profiler is active." << std::endl;  
 return 0; // Success  
}  
  
/\*\*  
 \* @brief The tool finalization function.  
 \* This is called by the SDK on application exit.  
 \*/  
void tool\_fini(void\* tool\_data)  
{  
 tool\_state\_t\* state = static\_cast<tool\_state\_t\*>(tool\_data);  
  
 std::lock\_guard<std::mutex> lock(g\_print\_mutex);  
 std::cout << " tool\_fini starting... Cleaning up." << std::endl;  
  
 // Must stop context before destroying  
 ROCPROFILER\_CALL(rocprofiler\_stop\_context(state->ctx), "context stop failed");  
  
 // Clean up all allocated SDK resources  
 ROCPROFILER\_CALL(rocprofiler\_destroy\_counter\_config(state->profile),  
 "profile destroy failed");  
 ROCPROFILER\_CALL(rocprofiler\_destroy\_buffer(state->buff),  
 "buffer destroy failed");  
 ROCPROFILER\_CALL(rocprofiler\_destroy\_context(state->ctx),  
 "context destroy failed");  
  
 // Clean up our state object  
 delete state;  
  
 std::cout << " tool\_fini complete." << std::endl;  
}  
  
  
// --- The "Magic" Entry Point ---------------------------------------------- //  
  
/\*\*  
 \* @brief The mandatory handshake function.[4, 6]  
 \* This is the \*only\* function the tool must export.  
 \* It is called by rocprofiler-register during runtime initialization.  
 \*/  
extern "C" rocprofiler\_tool\_configure\_result\_t\* rocprofiler\_configure(  
 uint32\_t version,  
 const char\* runtime\_version,  
 uint32\_t priority,  
 rocprofiler\_client\_id\_t\* client\_id)  
{  
 (void) version; (void) runtime\_version; (void) priority;  
  
 // Set the client name  
 client\_id->name = "MyCounterTool";  
  
 // Allocate the persistent state object   
 tool\_state\_t\* state = new tool\_state\_t{};  
 state->client\_id = client\_id;  
  
 // Define the tool's configuration  
 static rocprofiler\_tool\_configure\_result\_t config = {  
 sizeof(rocprofiler\_tool\_configure\_result\_t),  
 tool\_init, // Pointer to our init function  
 tool\_fini, // Pointer to our fini function  
 static\_cast<void\*>(state) // Pass our state object as tool\_data  
 };  
  
 // Return the configuration to the SDK  
 return &config;  
}

## Section 5: Building and Executing the Custom Profiler

### 5.1 The Build System: A Complete CMakeLists.txt

The application (main.cpp) and the tool (tool.cpp) must be compiled as two separate targets. The application is a HIP executable, while the tool is a standard C++ shared library that links against the rocprofiler-sdk.

This CMakeLists.txt will build both targets correctly, assuming ROCm is installed in /opt/rocm.

CMake

# CMakeLists.txt  
cmake\_minimum\_required(VERSION 3.21)  
project(RocprofilerExample CXX HIP)  
  
# Find the ROCm installation  
set(ROCM\_PATH "/opt/rocm" CACHE PATH "Path to ROCm installation")  
set(CMAKE\_PREFIX\_PATH ${ROCM\_PATH})  
  
# Find the HIP package  
find\_package(hip REQUIRED)  
  
# -----------------------------------------------------------------  
# Target 1: The HIP Application (my\_app)  
# -----------------------------------------------------------------  
add\_executable(my\_app main.cpp)  
  
target\_link\_libraries(my\_app PRIVATE hip::host)  
  
# -----------------------------------------------------------------  
# Target 2: The Profiler Tool (my\_tool)  
# This is a C++ shared library, NOT a HIP library.  
# -----------------------------------------------------------------  
add\_library(my\_tool SHARED tool.cpp)  
  
# Add include directories for rocprofiler-sdk  
target\_include\_directories(my\_tool PRIVATE ${ROCM\_PATH}/include)  
  
# Link against the rocprofiler-sdk runtime library  
target\_link\_libraries(my\_tool PRIVATE ${ROCM\_PATH}/lib/librocprofiler-sdk-runtime.so)  
  
# -----------------------------------------------------------------  
# Install step (optional)  
# -----------------------------------------------------------------  
install(TARGETS my\_app my\_tool DESTINATION bin)

**To build using CMake:**

Bash

$ mkdir build  
$ cd build  
$ cmake..  
$ make

### 5.2 Manual Compilation (g++ / hipcc)

For those preferring manual compilation, the following commands achieve the same result.

**1. Compile the HIP Application:**

Bash

$ /opt/rocm/bin/hipcc main.cpp -o my\_app

**2. Compile the Profiler Tool (as a shared library):**

Bash

$ g++ -shared -fPIC tool.cpp -o libmytool.so \  
 -I/opt/rocm/include \  
 -L/opt/rocm/lib -lrocprofiler-sdk-runtime

### 5.3 Execution: Putting It All Together

To run the application with the tool, the ROCP\_TOOL\_LIBRARIES environment variable must be set to point to the compiled libmytool.so file.4

Bash

# Ensure the shared library is in the current directory or library path  
$ export ROCP\_TOOL\_LIBRARIES=./libmytool.so  
  
# Run the application  
$./my\_app

**Expected Output:**

The output will be an interleaving of std::cout statements from both the application ([App]) and the tool's callbacks (``).

tool\_init starting...  
 Found GPU agent: gfx90a (ID: 12345)  
 Found requested counter: SQ\_WAVES  
 Found requested counter: TCC\_HIT  
 tool\_init successful. Profiler is active.  
[App] Launching vector\_add kernel...  
 Dispatch Callback: Intercepted kernel 'vector\_add'. Applying counter profile.  
[App] Kernel launch complete.  
[App] Verification: PASSED  
[App] Shutting down in 1 second...  
 tool\_fini starting... Cleaning up.  
  
 BUFFER CALLBACK: Received 2 records.  
 > DISPATCH HEADER:  
 Correlation ID: 1  
 Kernel Name: vector\_add  
 > COUNTER VALUE:  
 Counter ID: 50123  
 Name: SQ\_WAVES  
 Value: 1024  
 > COUNTER VALUE:  
 Counter ID: 40567  
 Name: TCC\_HIT  
 Value: 64  
 BUFFER CALLBACK: Finished processing records.  
  
 tool\_fini complete.

## Section 6: Reference: Key Hardware Counters and Data Interpretation

### 6.1 How to Find Available Counters (Programmatically)

The list of available counters is specific to each GPU architecture (e.g., MI200 vs. MI300).23 A simple tool can be created to dump all available counters by modifying the tool\_init and counter\_iterate\_cb from the sample above to print *all* counter names, not just the requested ones.

### 6.2 Table 1: Key Hardware Performance Counters

The following table lists a "starter set" of useful counters for performance analysis, primarily drawn from the MI200/MI300 architecture documentation.2

| **Counter Name** | **Unit** | **Definition** |
| --- | --- | --- |
| SQ\_CYCLES | Cycles | Clock cycles. |
| SQ\_WAVES | Waves | Number of waves executed. 2 |
| SQ\_INSTS\_VALU | Instructions | Number of Vector ALU (VALU) instructions issued. 25 |
| SQ\_INSTS\_SALU | Instructions | Number of Scalar ALU (SALU) instructions issued. 25 |
| TCC\_HIT | Count | Number of L2 cache hits. (Instanced, see 6.3) 22 |
| TCC\_MISS | Count | Number of L2 cache misses. (Instanced, see 6.3) 22 |
| GRBM\_GUI\_ACTIVE | Cycles | Number of cycles the Graphics Register Bus Management (GRBM) is active. 2 |
| SQ\_VALU\_MFMA\_BUSY\_CYCLES | Cycles | Number of cycles the Matrix Fused Multiply-Add (MFMA) ALU is busy. |
| CPC\_ME1\_BUSY\_FOR\_PACKET\_DECODE | Cycles | Number of cycles the Command Processor (CP) Micro Engine (ME1) is busy decoding packets. |

### 6.3 The Instancing Problem: TCC\_HIT vs. TCC\_HIT[<instance>]

A critical pitfall for new users is the concept of *instanced counters*. Many hardware blocks, such as the L2 (TCC) cache, are not monolithic but are distributed across multiple independent banks or instances.22

A user who requests the "base" counter name TCC\_HIT (or TCC\_MISS) will receive an "unsupported" error during profile creation.21 This is because the name is ambiguous. The profiler must be told *which* instance to measure.

The rocprofiler\_iterate\_agent\_supported\_counters callback will expose the correct, instanced names (e.g., TCC\_HIT, TCC\_HIT,... TCC\_HIT). A tool must either:

1. Request one or more specific instanced counters (as done in the sample, TCC\_HIT).
2. Request a *derived, summed metric* (e.g., TCC\_HIT\_sum), which the profiler understands as an instruction to collect from all instances and sum the results.22

### 6.4 Interpreting Dimensional Data

The rocprofiler\_counter\_record\_t struct, which delivers the counter value, contains fields to identify the "Dimension" of the hardware block that produced the value (e.g., ROCPROFILER\_DIMENSION\_XCC, ROCPROFILER\_DIMENSION\_SHADER\_ENGINE).16 A single kernel dispatch on a multi-chip GPU will result in counter values being reported from multiple dimensions (e.g., one set of SQ\_WAVES per Shader Engine). A production-grade tool would parse this dimensional data to provide a granular, hardware-block-level view of performance.

## Section 7: Conclusion

This report has deconstructed the rocprofiler-sdk tool architecture, providing a first-principles guide to building a custom C++ performance counter tool. By moving beyond the rocprofv3 command-line utility, developers and researchers can unlock the full, fine-grained power of the ROCm profiling infrastructure.

The core takeaways are:

1. **Architecture:** The rocprofiler-sdk uses an **inversion-of-control** model. The tool is a passive shared library (.so) that is loaded by the ROCm runtime via the ROCP\_TOOL\_LIBRARIES environment variable 4 and activated through the exported rocprofiler\_configure symbol.4
2. **State Management:** A C-to-C++ state management pattern, using the void\* tool\_data pointer provided by the C API, is essential for building an object-oriented tool.8
3. **Critical Pitfalls:** Developers must avoid two key pitfalls:
   * **The tool\_init Deadlock:** Calling any HIP/HSA API from within tool\_init will cause a deadlock, as the runtime is not yet fully initialized.3
   * **Dispatch Counting Serialization:** Using the "Dispatch Counting" service, while necessary for per-kernel data, serializes all kernel execution, fundamentally altering application performance and potentially causing deadlocks in co-dependent applications.16
   * **Counter Instancing:** Base counter names like TCC\_HIT are ambiguous. The instanced (e.g., TCC\_HIT) or summed (e.g., TCC\_HIT\_sum) versions must be used.21

By synthesizing the fragmented API documentation into the complete, annotated code sample provided in Section 4, this report delivers the "missing" example 3 required to begin custom tool development. This foundation enables the creation of sophisticated, in-situ profiling solutions capable of conditional activation, dynamic filtering, and deep integration with complex HPC applications.

#### Works cited

1. AMD Profiling Tools for Performance Timelines - HLRS, accessed November 15, 2025, <https://fs.hlrs.de/projects/par/events/2024/GPU-AMD2/day1/03_AMD_profiling_tools.pdf>
2. Using rocprofv3 — ROCprofiler-SDK 1.0.0 Documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/how-to/using-rocprofv3.html>
3. ROCprofiler-SDK 1.0.0 Documentation - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/>
4. ROCprofiler-SDK tool library - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/api-reference/tool_library.html>
5. Introduction to ROC-Profiler (rocprof), accessed November 15, 2025, <https://462000265.lumidata.eu/4day-20241028/files/LUMI-4day-20241028-3_09_Introduction_to_Rocprof_Profiling_Tool.pdf>
6. August 2025 - package-review - Fedora mailing-lists, accessed November 15, 2025, <https://lists.fedorahosted.org/archives/list/package-review@lists.fedoraproject.org/2025/8/?count=200&page=1>
7. ROCm/rocprofiler-register: [DEPRECATED] Moved to ROCm/rocm-systems repo - GitHub, accessed November 15, 2025, <https://github.com/ROCm/rocprofiler-register>
8. ROCprofiler-SDK developer API: ROCProfiler-SDK API Specification ..., accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/_doxygen/rocprofiler-sdk/html/>
9. ROCProfiler with rocprof might fail to initialize in some PyTorch applications #4613 - GitHub, accessed November 15, 2025, <https://github.com/ROCm/ROCm/issues/4613>
10. ROC Profiler Library Specification - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler/en/docs-5.0.2/rocprofiler_spec.html>
11. ROCm consolidated changelog, accessed November 15, 2025, <https://rocm.docs.amd.com/en/latest/release/changelog.html>
12. Comparing ROCprofiler-SDK to other ROCm profiling tools, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/conceptual/comparing-with-legacy-tools.html>
13. Using rocprofv3 — ROCprofiler-SDK 0.5.0 Documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/docs-6.3.1/how-to/using-rocprofv3.html>
14. roctracer matrixtransport\_test example can not run with rocprof · Issue #1533 - GitHub, accessed November 15, 2025, <https://github.com/ROCm/ROCm/issues/1533>
15. Rocprofiler SDK - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/_/downloads/rocprofiler-sdk/en/docs-6.3.0/pdf/>
16. ROCprofiler-SDK counter collection services - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/api-reference/counter_collection_services.html>
17. ROCm 6.3.0 Release #4107 - GitHub, accessed November 15, 2025, <https://github.com/ROCm/ROCm/discussions/4107>
18. ROCprofiler-SDK counter collection services - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/docs-6.3.2/api-reference/counter_collection_services.html>
19. Definitions — ROCprofiler-SDK 0.4.0 Documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/docs-6.2.1/api-reference/counter_collection_services.html>
20. Rocprofiler SDK - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/_/downloads/rocprofiler-sdk/en/docs-6.4.3/pdf/>
21. A consistent terminology for describing the basic performance counters (e.g. TCC\_HIT) in the ROCm documentation · Issue #116 · ROCm/rocprofiler - GitHub, accessed November 15, 2025, <https://github.com/ROCm/rocprofiler/issues/116>
22. Zero TCC\_HIT\_sum all the time · Issue #150 · ROCm/rocprofiler - GitHub, accessed November 15, 2025, <https://github.com/ROCm/rocprofiler/issues/150>
23. MI200 performance counters and metrics - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/en/docs-6.0.0/conceptual/gpu-arch/mi200-performance-counters.html>
24. MI300 and MI200 Series performance counters and metrics - AMD ROCm documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/en/latest/conceptual/gpu-arch/mi300-mi200-performance-counters.html>
25. rocprof — ROC Profiler Documentation, accessed November 15, 2025, <https://rocm.docs.amd.com/projects/rocprofiler/en/docs-5.1.3/>
26. rocprof or rocprofv2 or rocprofv3 :- Can an example to display the counters ie pmc : Wavefronts VALUInsts SALUInsts SFetchInsts be provided · ROCm rocprofiler-sdk · Discussion #38 - GitHub, accessed November 15, 2025, <https://github.com/ROCm/rocprofiler-sdk/discussions/38>